Part 1: Analyze Legacy Code

```
public DataTable GetCustomerInfo(string id)
{
    var dt = new DataTable();
    using (var conn = new SqlConnection("...")) // Connection string is l
    {
        conn.Open();
        var sql = "SELECT * FROM Customer WHERE id = '" + id + "'";
        using (var da = new SqlDataAdapter(sql, conn))
        {
            da.Fill(dt);
        }
    }
    return dt;
}
```

Task 1: In your own words, what is the primary purpose of this function?

Answer: This function receives a customer ID to collect and return customer information in the database as a DataTable type.

Task 2: Identify at least three distinct problems with this implementation. Consider aspects such as security, maintainability, and performance.

Answer: P1. The line `using (var con = new SqlConnection("..."))` embeds the database connection string directly within the source code, which has a security risk and maintenance burden.

P2. The line `var sql = "SELECT * FROM Customer WHERE id = '" + id + "'";` directly concatenates the customer ID string into the SQL query. This is an extremely severe security vulnerability

P3. The code doesn't have mechanisms to handle exceptions that might occur during database operations

Task 3: For each problem identified, briefly propose a specific improvement.

Answer: P1. Store connection strings in external configuration files, such as app.config in .NET Framework. These files are designed to hold configuration data, including connection strings.
P2. Use Parameterized Queries instead of concatenating values. For example

```
public DataTable GetCustomerInfo(string id)
{
var sql = "SELECT * FROM Customer WHERE id = @id"; // @id is
a placeholder
using (var cmd = new SqlCommand(sql, conn))
 {
    cmd.Parameters.AddWithValue("@id", id); // Value is added
as a parameter
    using (var da = new SqlDataAdapter(cmd))
    {
        da.Fill(dt);
    }
  return dt;
 }
}
```

It's to separate the ID value and placeholders in the SQL query. The database then understands that the values are data, not executable code, and handles them safely.
P3. Implement Try-catch blocks around operations, so when something goes wrong, users can see the error message and prevent leaked information.
(In this part I use gemini to find the problem and how to fix it)

PART 2: Rewrite and Modernize

(In this part I use gemini to create function in python using sqlite3 libary)

PART 3: Extend With New Logic

```
// Add new argument for start_date and end_date which has
default as none
public DataTable GetCustomerInfo(string id)
{
var sql = "SELECT * FROM Customer WHERE id = @id"; // @id is
a placeholder

// Add if statement to check if start_date and end_date are
not none
// If both dates are provide add new placeholder for the date
arguments

using (var cmd = new SqlCommand(sql, conn))
 {

    cmd.Parameters.AddWithValue("@id", id); // Value is added
as a parameter
// Check if the both dates are not none then add value to the
new placeholder
    using (var da = new SqlDataAdapter(cmd))
    {
        da.Fill(dt);
    }
  return dt;
 }
}
```

(In this part I use gemini to create an extend functionality)

PART 4 (Optional): System and User Perspective

The user can retrieve the customers data by their ID and they can filter the customer information by created_date within start_date and end_date so when customer has many information rows the user can get the specific informations in that range.

After the searched the results would display info on customer card like : name, email, phone, created date

If the date filter has provide in the transaction would display: transaction detail, status

If not it would display message “Customer info not found”

NAKROP NARUN